GEM

A Generalised Event Monitoring Language for Distributed Systems

Masoud Mansouri-Samani and Morris Sloman

Imperial College Research Report No. DoC 95/8

(Revised version of Report No. DoC 93/49, December 93)

12 July 1995

Abstract

Event based monitoring is critical for managing and debugging networks and distributed systems. This paper presents GEM – an interpreted Generalised Event Monitoring language. It allows high level, abstract events to be specified in terms of combinations of lower level events from different nodes in a loosely coupled distributed system. Event monitoring components can thus be distributed within the system to perform filtering, correlation and notification of events close to where they occur and thus reduce network traffic. GEM is a declarative rule based language in which the notion of real time has been closely integrated and various temporal constraints can be specified for event composition. The paper discusses the effect of communication delays on composite event detection and presents a tree based solution for dealing with out-of-order event arrivals at event monitors.

Keywords:

Monitoring, distributed systems, event reporting, event correlation, event filtering, composite events.

Department of Computing
Imperial College of Science, Technology and Medicine,
180 Queen's Gate,
London SW7 2BZ,
UK

Email: mm5@doc.ic.ac.uk, mss@doc.ic.ac.uk

1 INTRODUCTION

Monitoring is essential for all aspects of management of communication networks and distributed systems. Managers receive status and event reports which are used to make management decisions which result in control actions being taken to modify the behaviour of the managed systems. Large systems may potentially generate large numbers of events, so it is necessary to filter and combine low-level events to form higher level ones in order to raise the level of abstraction for both human and automated managers and prevent them being swamped by event reports. The ability to combine events is also needed for detecting abnormal occurrences for debugging purposes during the development and testing phases of a system.

Certain inherent characteristics of distributed systems make their monitoring more complicated than centralised systems. A typical distributed system consists of a number of cooperating processes executing on different machines and communicating via message-passing. Components must be instrumented to generate the required information in the form of status and event reports. Lack of a global clock and variable and unbounded communication delays makes it difficult to correlate monitoring information in order to arrive at a consistent global view of system state. Centralised correlation of event reports is not practical in very large distributed systems.

The most commonly adopted approach for observing and analysing the behaviour of distributed systems has been *event-driven monitoring* (Bates 1995; Haban & Wybranietz 1990; Jakobson & Weissman 1995; LeBlanc & Robbins 1985; Lumpp et al. 1990; Spezialetti & Bernberg 1995; Spezialetti & Kearns 1988; Wolfson et al. 1991). System behaviour is monitored in terms of a set of *primitive* events representing the lowest observable system activity. Users specify *composite* events using composition operators relating to event sequences which may span events from several processes and *temporal constraints*. For example a composite event can be defined as event e1 followed by event e2 within a window of 5 seconds. A composite event is detected when the specified pattern of events is recognised but the variable event delays make detection of distributed composite events particularly challenging — how to detect a composite event which is not invalidated as a result of a late-arriving event to which it refers. Current systems either do not address this problem or provide limited flexibility to deal with it.

Temporal constraints are essential for many management activities in a distributed environment. For example a configuration manager must start a new server when an existing one fails and the automatic recovery system does not succeed to bring it back up again within 10 minutes of its failure. Delayed and unordered delivery of events representing server failure, successful recovery and the scheduled time event may result in a new service being created unnecessarily.

Another issue that needs more attention is the changing monitoring requirements. As the system evolves there may be a need to dynamically change the patterns of activity which is monitored without having to bring the operation of the system to a halt so there is a need to dynamically change event specifications. For efficiency purposes it would be best to perform monitoring activities as close to the source of event reports as possible. This stipulates a *dynamic and distributed event monitoring service.*

Figure 1 shows the general overview of an event service which is an integral part of a generalised distributed monitoring service (Monsouri-Samani & Sloman 1994). Event generators may be instrumented monitored objects or other components responsible for polling and observing the activity and changes in the status of monitored objects. Event monitors process event reports from different nodes in the system by performing filtering, composition and notification on these reports based on a given specification. Event disseminators send event reports to clients who subscribe to receive them. Note that event monitors can themselves act as generators for other monitors.

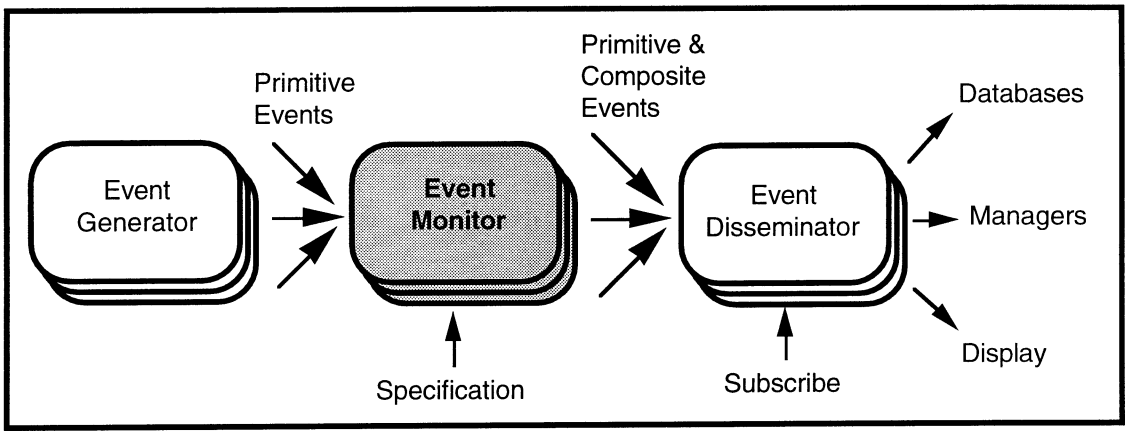

Figure 1 Components of an event monitoring service

In this paper we describe GEM which is a declarative and interpreted language used for specifying the operation of event monitors. Each monitor contains a command interpreter, and can be controlled interactively by sending it the appropriate GEM scripts. A GEM script declares events classes, rules which define the actions to be taken when an event is triggered and commands to trigger events, disable or enable rules etc. GEM allows the programmer to specify arbitrarily complex (composite) events and guards in the LHS of the rules. We make no assumption about the order of occurrence and detection (arrival) of events. The user can specify a variety of temporal constraints and is able to deal with delays in a flexible and efficient manner using built-in features of the language. The only assumption that is made about time is the existence of a well-synchronised global clock[1].

In section 2 we shall describe how composite events can be specified. Section 3 explains the features of the GEM language followed by example GEM scripts in section 4. Section 5 describes the problems introduced by delays in detecting composite events and shows how it is dealt with in GEM. The implementation of an event evaluation tree together with a detailed example of the operation of the detection algorithm employed are presented in section 6. Related work is mentioned briefly in section 7.

2 EVENT SPECIFICATION

An *event* is a happening of interest which occurs instantaneously at a specific time. A specified event in GEM may be *primitive* or *composite*, and may have a *guard* associated with it. The nature and the semantics of such events depend largely on the application domain and the system that is being monitored. For management purposes certain events such as object creation, deletion and migration, each with appropriate attributes, may have to be monitored. In databases, events of interest may include reads, writes, commits and aborts. In GEM the user can dynamically add new event classes to the existing set of event types which can be detected by the monitor.

2.1 Primitive Event Specification

In its general form a primitive event is specified using a guarded primitive event expression:

```
guarded-primitive-event-expr:= primitive-event-expr [when guard]
```

The guard is an optional predicate which is evaluated only when the primitive event occurs and is used to mask or hide its occurrence. The absence of a guard is treated as one which always evaluates to true. Guards are further discussed in section 2.3. A primitive event forms

---

[1] The clocks on our machines are closely synchronised using the *probabilistic clock synchronisation algorithm* (Christian 1989).

the smallest building block from which more complex events may be constructed and it is specified as follows:

```
primitive-event-expr:=
        event-id | * | every <time-period-expr> | at <time-point-expr>
```

An *event-id* identifies the type or the class of an *explicit* user-declared event.

Attributes of an event carry information about the action that caused it to occur. Predicates on attribute values can be used as the basis for filtering, composition or to generate new events. *Event independent attributes* are common to all event classes and are maintained by the monitor:

- *event id.*: a unique event class identifier
- *source id.*: where the event occurred
- *timestamp* : the best estimation of the occurrence time of the event

Event dependent attributes are optional, are specific to an event class and require explicit declaration.

For an internally triggered event the identity of the monitor and the occurrence time are used as its source identifier and timestamp respectively. Its event dependent attributes are set to those supplied by the programmer. This is further described in sections 3.

An external event is triggered when the corresponding *event report* arrives at the monitor and received as a null terminated string of characters with the following format:

```
event-id [<source-id>][<timestamp>][(<attribute-value-list>)]
```

For example,

```
warning "sensor1" [10:30 27/6/95] (-4.5, "Temperature too low")
```

The event class identifier is compulsory for every report. The source-id may be omitted in which case the event is assumed to have been associated with an external entity. This is useful in circumstances when the source of the event is not important for monitoring purposes (e.g., an alarm message received from *any* sensor may be of interest). The *timestamp* is the occurrence time of the event based on the synchronised global clock. If the event generator was a simple device with no access to a clock, then the monitor inserts the detection time i.e. report arrival time as the best estimate of the occurrence time. An event report may also contain a list of values corresponding to event dependent attributes as described in section 3.1. The '*' operator is used to mean *any* user-declared event.

The *every* operator is used to specify a periodic time event. It is followed by a time period expression. The constants *day*, *week*, *month*, *year*, *hour*, *min*, *sec*, *msec* (milliseconds) represent the corresponding time periods. E.g.,

```
every[min]          /*every minute */
every[2*day]        /*every two days*/
every[day+12*hour]  /*every one and a half days*/
```

Such events are considered to be *implicit*, and need no explicit declaration before they are used. The start of the period is the time at which the corresponding rule is enabled. Specified time periods are automatically normalised (e.g., [65*sec] is changed to [1*min+5*sec]).

The *at* operator is used to specify an event at a specified time and date with the following format:

```
[hours:minutes:seconds.milliseconds weekday day/month/year]
```

Time of day and date components have the usual fields. Some of these fields may be omitted (e.g., if the last two fields of the time of day – seconds and milliseconds – are left out they are automatically set to zero). When specifying the time of day the first two fields (hours and minutes) are compulsory. The keywords *Mon*, *Tue*, *Wed*, ..., *Sun*, can be used to refer to the corresponding days of the week. In addition to the usual numerical values, the keywords *Jan*, *Feb*, *Mar*, ... , *Dec*, can be used to refer to the corresponding months. It is assumed that the monitor maintains the current time in a predefined global variable, *now* which is continually updated and can be referred to in the guards and rule actions.

A time point specification may be *exact* or *partial*. In general the exact specification represents an absolute temporal event (a single occurrence) whereas a partial specification may represent a *set* of such occurrences with respect to the current time[2]. The latter is obtained by omitting one or more of the fields of the time specification or by using a wild card character '-' to match any valid time for that field. When the rule is enabled and on every occurrence, the next possible time point matching the specification is calculated and is used to set the omitted fields of the specification. In some cases no such time point may be found (e.g., when the last possible time point matching the specification has already passed).

Table 1 gives some examples of time point specifications, their types and relevant explanations. The GEM interpreter automatically checks the validity of the time specifications and produces appropriate error messages if necessary (e.g., For `at[01:30 Thu 29/Feb/1996]`: Does February have 28 or 29 days in 1996? Is the specified date a Thursday?).

Time specification	Type	Explanation
`at[10:30]`	partial	Occurs at 10:30 am every day. Unbounded number of occurrences.
`at[Fri]`	partial	Occurs at time 0:0:0.0 every Friday. Unbounded number of occurrences.
`at[29/Feb/96]`	exact	Occurs at time 0:0:0.0 on 29th February 1996. The specification of the week day is not really necessary, as it is automatically derived and set from the specified date.
`at[01:30 29/Feb/96]`	exact	The same as above except that it would occur at 1:30am.
`at[01:30 Fri]`	partial	Occurs at time 1:30 am every Friday. Unbounded number of occurrences.
`at[01:30 -/Feb/96]`	partial	Occurs at 1:30 am every day in February 1996. Bounded number of occurrences.
`at[Fri 13/-/-]`	partial	Occurs at 0:0:0.0 am every Friday 13th. Unbounded number of occurrences.
`at[1/-/96]`	partial	Occurs at 0:0:0.0 am on the first day of every month in 1996. Bounded number of occurrences.

Table 1 Examples of time point specifications

2 Note that the number of these occurrences may or may not be finite. Also the maximum number of repetitions of such events is once a day. To restrict the number and the frequency of occurrences, the rule which it is a part of, has to be disabled as described later.

When a rule containing such a time specification is enabled, the monitor sends an appropriate request to a *timer* component. It updates the current time (at a rate set at configuration time) and notifies the monitor whenever a request can be satisfied.

2.2 Composite Event Specification

In its most general form an event is specified by using a guarded composite event expression:

```
guarded-composite-event-expr:= composite-event-expr [when guard]
```

Where a composite event expression is a combination of other primitive and composite event expressions, using various event specification operators.

Intuitively an event occurs at a single point in time. A composite event, however, may consist of a number of primitive events occurring at different times and therefore such an event is detected in the context of a history. The first primitive event which starts a new occurrence is called the *initiator* and the last one which causes the composite event to occur is called the *terminator*. Therefore we can define a time period, called the *occurrence* interval, which starts from the initiator at time t_1 and ends at the terminator at time t_n. It represents the period during which the event is *in the process of occurring* (i.e., t_n-t_1). Note that t_n is usually considered to be the *occurrence time* of the composite event. In GEM the timestamps of the initiator and the terminator of a composite event can be accessed by using the |@ and @ operators, respectively, as shown later.

When the event specified by the composite event expression occurs its guard is evaluated, and if true, the event is said to have occurred. The guard is optional and could refer to event attributes or global variables (e.g., *now*). Guards are discussed in section 2.3. Figure 2 shows the structure of a typical event expression.

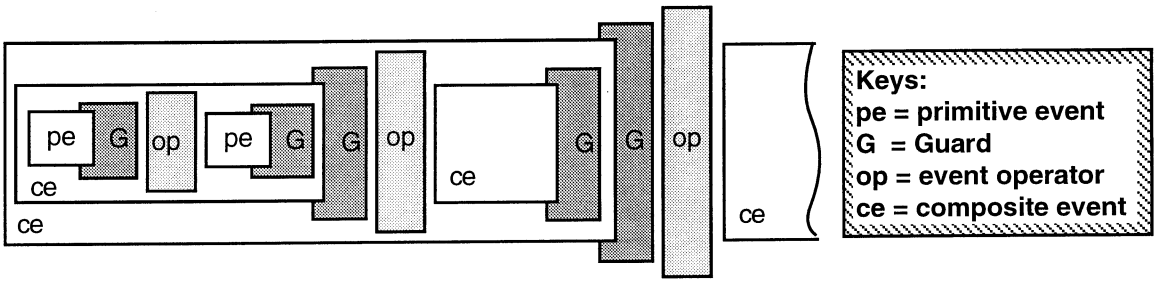

Figure 2 The structure of a typical composite event expression

Associating a guard with the constituent primitive events as well as a *global* guard associated with the whole composite event expression allows a greater control over event detection and provides a more powerful filtering facility.

Table 2 describes the event composition operators supported by GEM. Given the independent nature of the rules and the ability to enable and disable them and also associate guards with events not all these operators are absolutely necessary. The operators *&*, *!* and + can be considered as our basic event composition operators. Other operators can be defined in terms of those.

Operator	Explanation	Comments
`e1 & e2`	Occurs when *both* e1 and e2 occur irrespective of their order.	Commutative and associative
`e + time-period`	Defines a scheduled time point event which occurs a specified period of time after the occurrence of event *e*.	e.g. `(e1 & e2) + [1*day+12*hour]` occurs a day and a half after the occurrence of both *e1* and *e2*.
`{e1 ; e2} ! e3`	Occurs when *e1 occurs* followed by *e2* with no interleaving *e3 event*.	The non-occurrence of an event only makes sense in the context of a specified closed interval. *e3* is considered to be a *cancellation event*.
`e1 | e2`	Occurs when *e1* or *e2* occurs	Commutative and associative
`e1 ; e2`	Occurs when e1 occurs before e2 (i.e., e1's timestamp is less than that of e2).	Permits event sequences to be specified. It is shorthand for `(e1 & e2) when (@e1 < @e2)`

Table 2 Event composition operators

2.3 Guards

A guard is a boolean expression that is used to mask the occurrence of a primitive or composite event. It typically includes predefined predicates and C-like expressions involving event attributes . For example,

```
temp_update when temp_update.t > 10
```

The keyword *when* can be viewed as a logical *and* whose right operand (predicate) is evaluated only when its left operand (event) occurs. A *when* clause could only refer to the attributes of events with which it is associated or global variables such as *now* (current time) and *self* (the identity of the monitor). Arbitrarily nested *when* clauses can be written. E.g.,

```
((e1 when e1.a=<10) & (e2 when e2.b>20))
      when (e1.c==e2.d) && (e1.e>=e2.f)
```

Note the difference between the operators *&* and *&&* . The former is used as a composite event operator and implies no time constraints (events e1 and e2 may occur in any order) whereas the latter is used in guards and implies that both sub-expressions of the guard must be satisfied at the same time.

The *class* identifier of an event can be used to refer to an *instance* of that event but this does not work for sequences of events of the same class e.g. (e1;e1;e1).

In GEM an *event variable* may be associated with a primitive or composite event expression using the ':' operator. For example, in *x:(e1;e1;y:e1)*, x is associated with the whole composite event and is used as a local name for it inside the rule, and y represents the third occurrence of e1 in the sequence. The scope of an event variable is the current rule only. Event variables are similar to free variables in a logic programming language and are needed to:

- Refer to specific event instances and access their attributes when multiple references are made to the same event class;

- To coordinate the evaluation of various event subexpressions and allow the user to refer to the same *occurrence* of an event (and its attributes);
- Refer to the start and end time points of the occurrence interval of a composite event — the only attributes of a composite event that can be accessed.

A guard's sub-expressions are evaluated in the order that they are written. The evaluation of the conjunction of these subexpressions is suspended and returns false as soon as one of them evaluates to false. It is a good idea to evaluate the temporal constraints first so that event information that do not satisfy those conditions can be discarded as soon as possible.

Occurrence intervals

The passage of time is of critical importance in determining what happens in a distributed system (e.g., whether two activities are carried out in parallel). Conventional programming languages have no built-in notion of time. In GEM the notion of real-time has been integrated within the language, allowing the users to talk about absolute and relative time points and various intervals in terms of these time points. As discussed before, the occurrence interval of a (composite) event is the period of time during which it is in the process of occurring. In guards the start and end points of this interval can be accessed by using the operators |@ and @, respectively. For example, the following event,

```
x:(e1 & e2 & e3) when (@x - |@x) < [3*sec]
```

occurs when three events *e1*, *e2* and *e3* , occur (in any order) and their occurrence interval is less than three seconds.

Operators @ and |@ allow us to detect the temporal relationships of two composite event occurrences (e.g., overlapping or one occurred before the other). Figure 3 shows some important temporal constraints that can be checked as part of a guard for composite events with variables *x* and *y*, respectively.

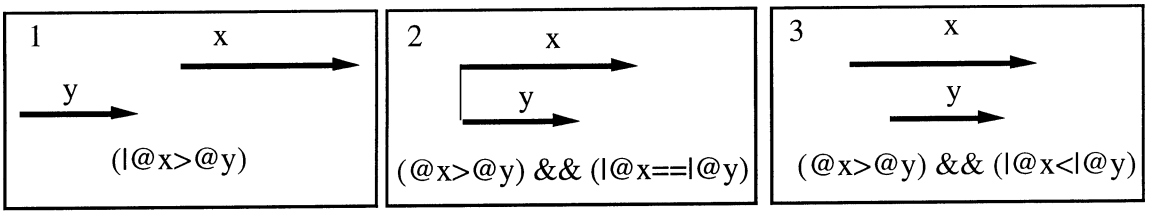

Figure 3 Some temporal constraints based on start and end time points

Note that, due to the way guards are evaluated, the following two event expressions, though similar, have different effects.

```
(i)   at[10:00] ; e1 ; at[12:00]

(ii)  e1 when ([10:00] < @e1 ) && ( @e1 < [12:00])
```

In (i) the composite event occurs at 12:00 when the sequence of specified events occur, whereas the event specified by (ii) occurs every time *e1* occurs between the two given time points and its occurrence timestamp is that of *e1*.

As it will be discussed later event information is maintained as long as it can potentially contribute to *new* rule firings and is not considered out of date, as specified by the user. Temporal constraints are particularly important because they can determine when to discard this information.

3 GEM SCRIPTS

A GEM script may consist of *event declarations*, *rule definitions* and *top level commands* and can be down-loaded to a particular event monitor.

3.1 Event Declarations

Event declaration is necessary to determine the explicit user-declared classes of events that can be manipulated by the monitor and the names and types of their attributes. An event declaration (for both internally and externally triggered events) has the following format:

```
event <event-id> [(<formal-attribute-declarations>)]
```

where *event-id* uniquely identifies the class of the event. For externally triggered events it provides an association between the monitoring report and the event triggered by the monitor when that report is received. The optional *formal-attribute-declarations* provides the names and types of event dependent attributes associated with the named event. Event independent attributes do not require explicit declaration.

The following types are supported in GEM: *char*, *long*, *double*, *string*, *time*, *itime*. The first four are similar to those in C++. The types *time* and *itime* are used for time point and time period expressions, respectively. E.g.,

```
event signal
event warning(double temp, itime interval, string mesg)
```

Special operators have been provided for accessing event independent attributes. For an event instance e, @*e* returns its timestamp and *$e* returns its source id. Event dependent attribute values can be accessed by using the usual '.' notation. Attribute *a* of an event instance *e* is accessed by *e.a.*

3.2 Rule Definitions

As mentioned before, the monitor is passive until an event is triggered. A rule definition specifies a sequence of *actions* to be performed upon the occurrence of an event and has the following format:

```
rule <rule-id> [<detection-window>]
   { <event-expression> ==> <action-sequence> }
```

Where,

rule-id	is the unique identifier used to refer to the rule,
detection-window	specifies the time window in which the rule operates,
event-expression	specifies the composite event to be detected, and
action-sequence	specifies the actions to be executed when the rule fires.

Each time an event is triggered the monitor tries to satisfy as many rules as possible (i.e., the event is "seen" by all rules which are dependent on it). When the event specified in the LHS of a rule is detected, the rule fires and its action sequence is executed. As we shall see, the firing of a rule can cause other rules to fire, by explicit triggering actions. We have adopted a simple execution semantics for rules. All the rules that can fire will fire and their action sequences are executed to the end. Other more sophisticated execution semantics exists in rule based languages such as OPS5 (Brownston 1985). In OPS5 a conflict resolution algorithm is used based on factors such as specificity and condition counts in the LHS of the rules to decide which one of the rules must be chosen for execution.

3.2.1 Detection Window

When the monitor is instantiated a default *detection window* (*dw*) from `now-dw` to `now` is specified that determines the sliding time window within which the monitor and therefore the rules may operate. The default can be overridden by defining a rule specific *dw* using a time period expression. Periodically the *dw* of all rules are moved forward in time by a predefined amount — a stepping time period[3], again set at monitor instantiation time. The occurrence time of an event that can potentially contribute to firing of the rule must fall within the rule's detection window, otherwise the event is ignored or discarded. In effect each rule operates on items (events) placed on a *conveyor belt*. These items fall off the edge at the end of the specified period of time, as shown in figure 4. The detection window has a dual purpose. It is used as a primary mechanism for dealing with delays by ignoring late arrivals and as a means of specifying persistency for events within the system. Each rule may maintain its own event history (as described in section 6). As the window moves forward, the portion of this history which falls outside dw is discarded.

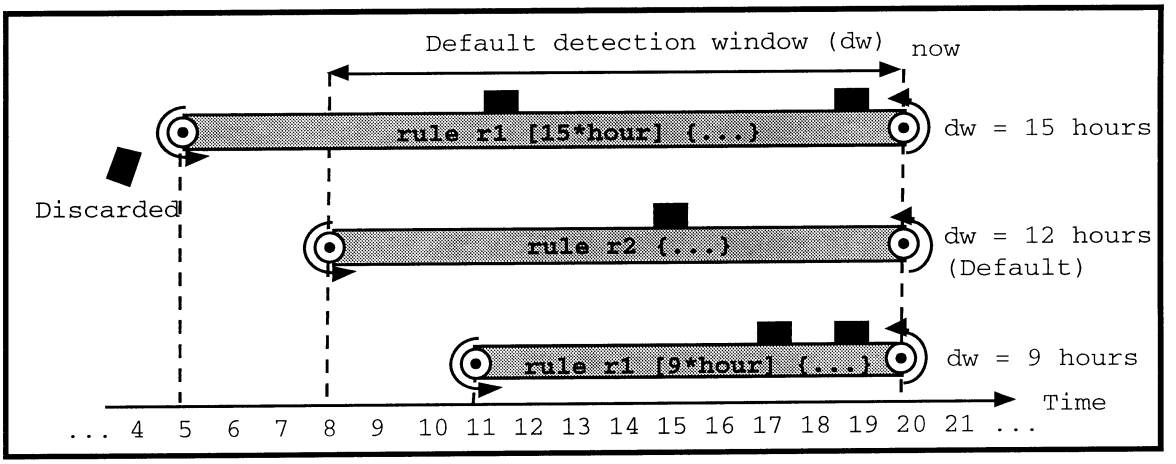

Figure 4 Conveyor belt analogy of a detection window

Intuitively *dw* indicates how long the programmer is interested in an event and enables the user to control the discarding of event information at rule level. This is particularly important when dealing with long delays and to avoid the constituents of an as yet incomplete composite event occurrence to linger on in the system forever.

3.2.2 Event Expression

Specification of events and the structure of event expressions were discussed at length in section 2. Section 6 describes the mechanism used for evaluation of such expressions.

3.2.3 Action Sequence

One or more actions can be specified in the RHS of a rule which are executed in the form of a procedural code. The following actions are possible: *notify*, *forward*, *trigger*, *enable* and *disable*.

a) *notify* Action

The *notify* action can be used to explicitly generate and report a new instance of a user-declared event. It has the following format:

```
notify <event-id> [(<attribute-value-list>)]
```

---

[3] In selecting the value of the stepping time period the clock resolution, the expected rate of event occurrences and the available size of the memory may have to be taken into account.

where,

event-id	specifies the class of the event to be notified, and
attribute-value-list	is the list of values for its event dependent attributes

The position, type and number of the supplied and expected attribute values must match exactly (c.f., parameter passing in C). Its timestamp is set to the occurrence time of the composite event in the LHS of the rule and its source id. is set to the monitor's own identifier. The newly generated event would not be visible internally (i.e., other rules will not see it). Only "the outside world" will detect its occurrence. For example, given the following event declaration:

```
event e1
event e2(long l)
event e3(long l, double d, string s)
```

we can write:

```
rule testrule {\
      <an arbitrary event expression> ==>\
            notify e1;\
            notify e2(10);\
            notify e3(10, 3.4, "Hello World")\
}
```

As shown in this example notifications with varying levels of details can be generated for different managers.

b) *forward* Action

The *forward* action is used to report one or more user-declared primitive event occurrences which contributed to the rule firing and is particularly useful for filtering purposes. This differs from the *notify* action in that *forward* reports one or more *existing* primitive event instances whereas *notify* effectively causes a new event instance (with possibly new timestamp, source id. and user specified attributes) to be generated and reported. This action has the following format:

```
forward (<event-id>|<event-variable>)
```

An event class identifier can be used to directly refer to the corresponding primitive event occurrence. For example, given a predeclared event, `link_failure`, we can write:

```
rule failure { link_failure when ... ==> forward link_failure }
```

An event variable can also be used to rewrite the `failure` rule in the following more compact form:

```
rule failure { x:link_failure when ... ==> forward x }
```

An event variable is necessary to unambiguously distinguish between multiple instances of an event class which may have to be reported. It also provide local naming for a composite event occurrence and is used to report its constituent user-declared primitive events. For example, given the predeclared events `e1` and `e2`, the following rules can be defined:

```
rule R1 { e1 ; e2 ==> forward e2 }
```

R1 reports the occurrence of an e2 after an e1 has occurred.

```
rule R2 { e1 ; e2 ; x:e2 ==> forward x }
```

R2 reports the second occurrence of an e2 after an e1 has occurred.

```
rule R3 { e1 ; x:(e2 & e3) ==> forward x }
```

R3 reports the occurrences of an e2 and an e3 (in any order), after an e1 has occurred.

c) *trigger* Action

A user-declared event can be triggered by simply "calling" it, much in the same way as a function call in conventional languages, with the following format:

```
<event-id> [(<attribute-value-list>)]
```

where,

event-id	specifies the class of the event to be triggered, and
attribute-value-list	is the list of values for its event dependent attributes

This is similar to the *notify* with respect to setting attributes. For example, given the following event declaration:

```
event signal_loss(long a, string m)
```

We can write:

```
rule r1 { <some composite event> ==> signal_loss(20, "Link Failure")
```

The newly triggered event is visible to all the other enabled rules within the monitor. The "outside world" cannot detect its occurrence unless explicitly reported by another rule using the notify or forward action. The trigger action provides a mechanism for easily naming and referring to the same composite event in different rules. E.g.,

```
rule r2 { ...; signal_loss;... ==> ...)
```

This provides a convenient abstraction mechanism. An internal event can be defined, triggered and notified in terms of other internal and external events and would typically inherit attributes from the events referenced in the LHS of the rule. An event triggering could in turn cause a number of rules to fire and other events to be triggered. The order in which the rules are fired is implementation dependent, but all the rules which can fire, will fire. A rule can not directly "see" the events that it triggers. E.g.,

```
rule r2 { ... ; signal_loss ; ... ==>  ...; signal_loss ; ...)
```

This prevents recursive firing of *r2*, given that no other rule makes *signal_loss* visible to it. Some event dependency checks may have to be performed to identify and avoid recursive and infinite triggering of events, and firing of rules. Circular event dependencies may span several rules, and the programmer is responsible for making these checks.

d) *enable* & *disable* Actions

GEM rules may be enabled or disabled using the following actions, respectively:

```
enable [<rule-id>]
disable [<rule-id>]
```

where, *rule-id* specifies the identity of the rule. If it is omitted, the identity of the current rule is used.

When an event is triggered only the enabled rules are examined. All rules are disabled by default and have to be enabled explicitly. Once enabled, a rule will remain so until it is specifically disabled. As we shall see the event evaluation tree of each rule may maintain historical data which may ultimately contribute to its firing. This data is automatically discarded when the rule is disabled. These actions provide a powerful mechanism for concentrating on specific aspects of system activity under certain conditions and filtering unwanted information. For example, enabling all the rules related to configuration and performance monitoring and disabling all the fault monitoring rules during a specific period.

In the following example, rules A and B refer to the user-declared events e1 and e2.

```
rule A { e1 ==> forward e1; e2; disable }
rule B { e2 + [5*sec] ==> enable A }
enable A
enable B
```

The effect of these rules and the top level *enable* command (as described later) is to report the occurrence of an event of type *e1* and to ignore all other occurrences (at least) for the next five seconds. This is particularly useful in fault management area and in particular the alarm correlation application in which a large number of similar alarms may be generated as a result of the same fault. All the duplicates can be removed for a specified length of time.

3.3 Top level Commands

GEM provides a number of top level commands which are independent of rules and are used for controlling the operation of the monitor. Some of these commands are similar to actions that can be performed in the RHS of the rules.

triggering events

As we described in section 3.2 an event can be triggered in the RHS of a rule. A similar effect can be produced at the top level by simply "calling" the event with appropriate attributes. The format is similar to that of an event report described in section 2.1:

```
event-id [<source-id>][<timestamp>][(<attribute-value-list>)]
```

This is useful for simulation purposes, checking event dependencies, or initialising the state of the event evaluation tree. For example, given these event declarations:

```
event e1 ( long l, string s, itime it )
event e2
```

the following can be used as valid top level event triggering commands:

```
e2
e2 [8:30 15/5/95]
e1 (3, "some string", [5*hour+30*min])
e1 [10:30:0.0 Mon 15/5/95] (3, "some string", [5*hour+30*min])
e1 "s10" [10:30 15/5/95] (3, "some string", [5*hour+30*min])
```

enable & *disable* commands

The *enable* and *disable* are similar to the corresponding actions described in section 3.2-(iii)-(d). They can be used to activate or deactivate rules. The format is the same with the exception that the wild card character '*' can only be used at the top level to enable or disable *all* the existing rules. e.g., `disable *`

stop command

The *stop* command is used to halt the operation of the monitor. It results in the discarding of all the event information held by the monitor.

~ (remove) command

As monitoring requirements change the set of declared event classes and defined rules will have to change. The remove command '~' allows us to delete those which are no longer needed. An event class can be removed using a command of the following form:

```
~ <event-id>
```

The specified event class can only be removed (undeclared) if there are no rules which specifically refer to it. If there exist rules dependent on the event class, an error message is displayed and the command is ignored. A rule can be removed using the following command:

```
~ rule <rule-id>
```

As a result of this command the event information maintained by the rule is discarded and the monitor removes the rule. Note that the removed rule may have been explicitly referenced by others. It is the user's responsibility to make sure that those rules are removed or changed accordingly.

load command

The *load* command can be used to instruct the monitor to read GEM commands from a specified file. E.g., `load "testfile.mi".`

4 EXAMPLES OF GEM SCRIPTS

Example 1

```
event power_cut
event disaster
```

the following rule causes a *disaster* report to be generated when the occurrence of a *power_cut* event is detected whose timestamp is between 24th December 1995 and 2nd January 1996 and whose source is the main generator.

```
rule disaster [10*min] {\
      x:power_cut\
            when ([24/Dec/95] < @x ) && ( @x < [2/Jan/96])\
                  ($x=="master generator") &&\
                  ==> notify disaster\
}
enable disaster
```

Note that rules and events may have the same names. This reduces the number of names in the system and allows the user to implicitly link a rule to the event it notifies or triggers. The detection window of the rule is set to 10 minutes. The *$* operator is used to access the source identifier of the event.

Example 2

```
event shut_down ( string reason)
```

The following rule can be used to generate a *shut_down* notification to the system administrator to shut down all the machines in order to protect them from the Friday 13th virus! The event expression is a partial time point specification. The keyword *at* can be omitted when specifying a time point, because it is used only as a syntactic sugaring.

```
rule protection { [Fri 13/-/-] ==>\
      notify shut_down("Friday 13th")}
enable protection
```

Example 3

Rule A reports the occurrence of the first user-declared event which occurs within 5 seconds of an event e1. The operator '*' is used as a wild card to refer to *any* user-declared event. Note that only event independent attributes (timestamp and source identifier) of such an event can be referenced in a guard. No detection window is specified for the rules. In this case it automatically defaults to the detection window specified at monitor start up time.

```
event e1
event e2 ( ... )
...
rule A { ( e1 ; y:*) when (@y-@e1) <= [5*sec] ==> forward y }
```

Example 4

The following script specifies that a gauge threshold notification must be generated when the value of the *val* attribute of a *gauge_changed* event crosses certain low or high threshold as shown in figure 5. The upper and lower thresholds are 10 and 5, respectively. The interval between these thresholds is called the *hysteresis interval*. The gauge value may oscillate about either of the thresholds and consequently multiple notifications may be generated. The *enable* and *disable* actions in the rules are used to force such notifications to be generated alternately when the value crosses the thresholds.

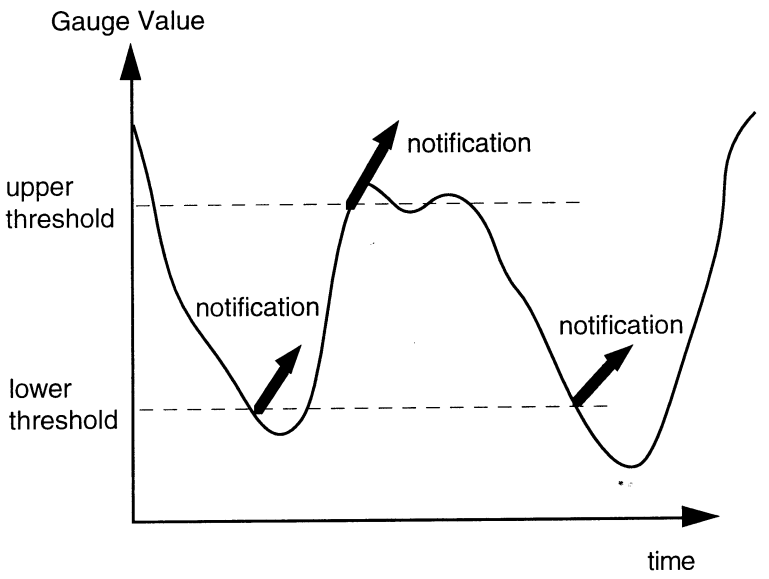

Figure 5 Gauge threshold operation

```
event gauge_changed( double val)
event upper_exceeded
```

```
event lower_exceeded
rule gauge_rule1 {\
   x:gauge_changed when x.val >= 10 ==> \
         notify upper_exceeded; enable gauge_rule2; disable }
rule gauge_rule2 {\
   x:gauge_changed when x.val <= 5 ==>\
         notify lower_exceeded; enable gauge_rule1; disable }
enable gauge_rule2
```

Example 5

The following *backup* rule specifies that a *backup* notification must be generated every hour between 8 am and 5 pm, on working days, and every two hours between 10 am and 4 pm during weekends. Such an event can be used for example by a system administrator to know when to make backups of all the system files. The *back_up* event depends purely on temporal events and guards not on any external events.

```
event backup
rule backup {\
    ( x:every[hour]\
         when ([8:00] < @x) && (@x < [17:00]) &&\
               ([Mon] <= @x) && (@x <= [Fri])) |\
   ( y:every[2*hour]\
         when ([10:00] < @y) && (@y < [16:00]) &&\
               ([Sat] == @y) && (@y == [Sun]))\
               ==> notify backup\
}
enable backup
```

Example 6

The following rule causes a *warning* event to be triggered internally for a pair of (temperature, pressure) alarms if both are received from the same source and are generated within 10 seconds of one another. The composite event occurrences are detected in chronological order given a maximum expected delay of 15 seconds.

```
event temp_alarm(double t)
event press_alarm(double p)
event warning(string sensor_id; double t; double p)
rule warning [15*sec] {\
((x:temp_alarm & y:press_alarm)\
   when (( $x==$y ) && (( @x-@y )< [10*sec] ))) + [5*sec]\
         ==> warning($x, x.t, y.p)\
}
enable warning
```

The detection window of 15 seconds takes into account the maximum delay and the temporal constraint specified in the guard. Although the events specified with variables x and y may occur in any order, the interval specified by @x-@y is always positive, no matter which one occurs first.

The scheduled time event represents the period of time that the rule must wait after the detection of the sub-event before it can fire. During this interval if an event arrives which chronologically precedes one of the existing pairs of events and a successful match can be found, it is automatically picked up instead by the algorithm. Therefore the older matching pair is correctly recognised as a composite event given a maximum delay of 5 seconds.

The event dependent attributes of *warning* event are set using the attributes of the sub-events in the LHS (i.e., the source identifier, the *t* attribute of the *temp_alarm* event and the *p* attribute of the *press_alarm*).

Example 7

The following script is based on a slightly altered version of an example given in (Jakobson & Weissman 1995). The rule EXPECTED_ALARM_RULE detects that a carrier group alarm type "A" on a network element occurred and during the following 1 minute interval an expected carrier group alarm type "B" did not occur at the same network element. On detecting such an event, it is forwarded by the monitor.

```
event ALARM_A
event ALARM_B
rule EXPECTED_ALARM_RULE {\
      ({x:ALARM_A ; z:(y:ALARM_A + [1*min])} ! w:ALARM_B)\
            when (?z && (x == y)) || (?w && ($x == $w))\
                  ==> forward x \
}
enable EXPECTED_ALARM_RULE
```

The equality between event variables can be used to see if they refer to the same event occurrence(s). A guard associated with event expressions constructed using '|' or '*!*' event operators may refer to the attribute of an event which has not actually occurred. To cope with this situation the operator '*?*' can be used to check and see if the event has actually occurred before referring to its attributes. In the above example, events z or w may occur after x. This is in concept similar to checking that a pointer is not zero before using it in C/C++.

5 DEALING WITH DELAYS

Variable communication delays in a distributed environment can cause variable delays between the *occurrence* and the *detection* times of an event. In addition events may not be detected in the order of occurrence. Event composition techniques, commonly used in centralised system, may not detect certain composite events or may result in erroneous detections and consequently unwanted rule firings. For example, the sequence of event instances (e_1 e_3 e_2) may be detected by the monitor in a different order (e_1 e_2 e_3). Given the delayed sequence of events the following rule:

```
rule r1 { {e1 ; e2} ! e3 ==> ... }
```

will *erroneously* fire on e_2 and its action sequence is executed before the cancellation event e_3 arrives as shown in figure 6.

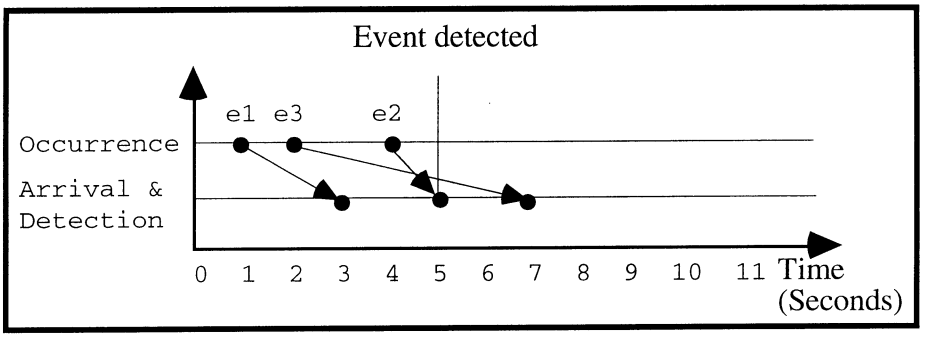

Figure 6 Ocurrence and detection of events

We call a detection *valid* when no late arriving event could invalidate or alter the detected composite event. Valid detection is impossible in the absence of explicit assumptions about the possible orderings or the maximum possible delays involved[4]. In many cases the best that can be done is to assume a maximum *tolerated* delay and to discard late arriving events.

5.1 Uniform Delaying Technique

Given a well-synchronised global clock and a known maximum possible communication delay D, one obvious approach which has been adopted by (Shim & Ramamoorthy 1990) is to uniformly delay *every* event by D time units before it is passed to the detection stage. This is illustrated in figure 7.

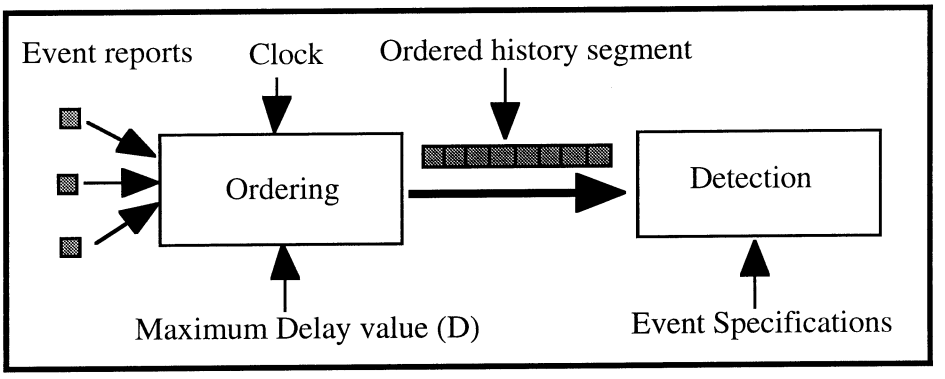

Figure 7 Uniform Delaying Technique

In the ordering stage every newly arrived event report e with occurrence timestamp t, is inserted in its correct place in a *global* ordered event history whose elements satisfy the following constraint: `(t + D) < C`, where C is the current time. On every clock update the segment of the history which has been delayed enough is forwarded to the detection stage where an existing composite event detection algorithm (e.g., based on petri-nets, evaluation trees or state machines) can be used to perform the detection. The application of this technique to our earlier example is shown in figure 8. D is assumed to be 6 seconds which is the time each event in the unordered sequence has been delayed. The ordered sequence can now be used for a valid event detection.

---

4 Even given certain guarantees about the maximum amount of delays, failures may prevent a valid detection.

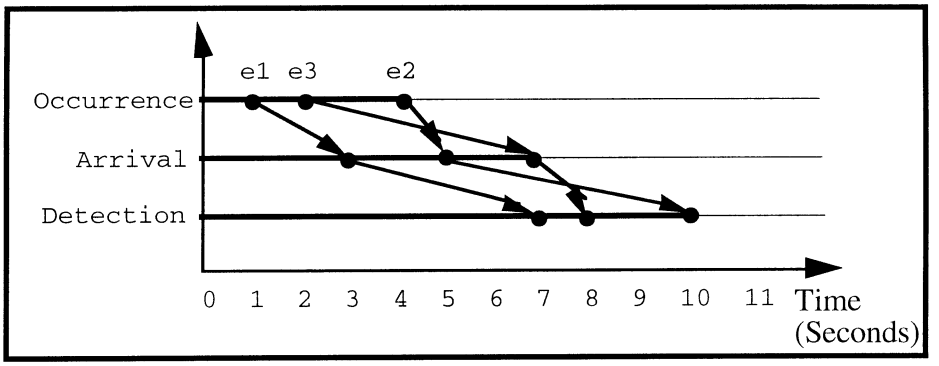

Figure 8 An example of the uniform delaying technique

This approach has several disadvantages. It is:

pessimistic – it introduces possibly unnecessary delays in event detection by keeping *all* the events by the maximum time delay,

inefficient – every event is ordered with respect to all the others in the maintained history even when temporal constraints are not important for that event in the ultimate detection stage,

inflexible – in some circumstances no explicit ordering may be necessary e.g. for events generated locally or in a local area network that provides ordering guarantees. This knowledge about specific events cannot be used.

5.2 Event Specific Delaying Technique

The approach that we have adopted is to push the knowledge and the capability of dealing with delays into the detection stage itself. In GEM a tree-based composite event detection algorithm is employed which makes no assumptions about communication delays or the order of arrival of event reports. Every new event is inserted in its correct place in an internal *hierarchical history*, maintained by an *event evaluation tree* (the structure of the tree and the history it maintains is described in section 6). The algorithm performs the ordering or imposes delays when needed and as specified by the user. In GEM delays are dealt with at two complementary levels:

i) A rule maintains an ordered hierarchical event history over a time span defined by the rule's detection window. A new event is inserted in its correct place in this history provided it is within the detection window, which represents the maximum *absolute* delay that the rule can cope with.

ii) A scheduled time event can be used as an explicit terminator event, to deliberately extend the detection interval. This value represents the maximum delay that can be accommodated *relative* to a specific event.

These can be viewed as absolute and relative *timeout* facilities. For example, given the previous example, with D=6 seconds, one can write a rule r2 as:

```
rule r2 { ({e1 ; e2} ! e3) + [6*sec] ==> ... }
```

Note that this rule uses a default detection window which must be greater than 6 seconds. Rule r2 can now cope with the sequence e_1 e_2 e_3 as shown in figure 9. At time 5 on the arrival of e2 the event specified by the sub-expression (`{e1 ; e2} ! e3`) is detected by the algorithm. This causes a relative time event to be scheduled for time 10 (e2's *occurrence* timestamp + delay). The information about the sub-event is maintained in the event evaluation tree while we wait for the scheduled event. At time 7 the cancellation event e_3 arrives. The algorithm makes sure that the sub-event and the scheduled time event are automatically cancelled so that the rule is not fired erroneously.

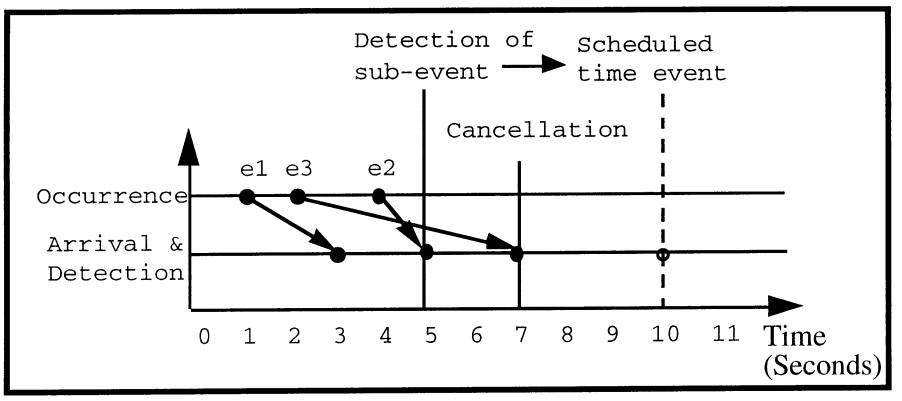

Figure 9 An example of event specific delaying technique

A scheduled time event can be associated with any event specified in GEM, and therefore a similar control over delays can be exercised at both primitive and composite event level. It provides the user with unlimited flexibility and a finer grain of control in dealing with delays[5]. This is particularly useful when events referenced in a rule may arrive from sources with different expected delays or where there is a known temporal relationship between event occurrences (e.g., events e1 and e2 will never occur more than 2 minutes apart). Furthermore the detection time is not affected by the external buffering strategy employed at a separate ordering stage.

6 IMPLEMENTATION

We have adopted a tree-based mechanism for composite event detection. It allows the event evaluation algorithm to deal with out-of-order sequences of event arrivals in an efficient manner.

6.1 Event Evaluation Tree

When a rule is interpreted, the event expression in its LHS is used to create an *event evaluation tree*, as shown in figure 10. The structure of the tree closely matches that of GEM's event expressions (figure 2) where the leaves and internal nodes represent guarded primitive and composite event expressions, respectively.

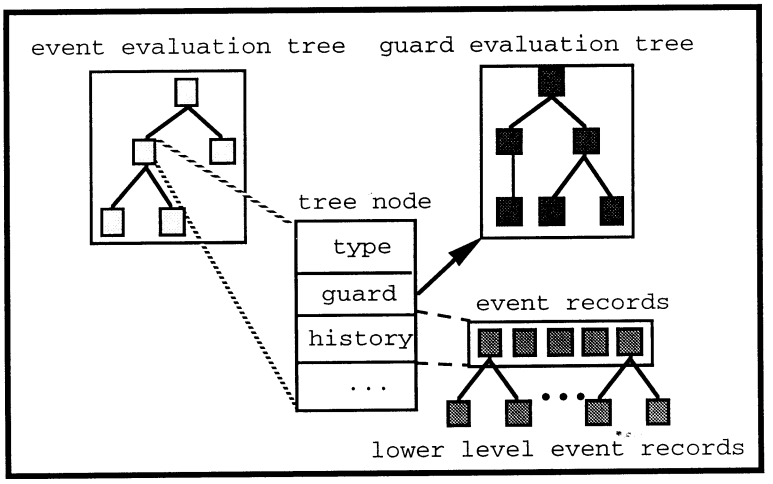

Figure 10 The structure of the event evaluation tree

---

5 It is not always possible to determine a maximum value for delays therefore the specified time period may represent the *tolerated* delay interval rather than an *expected* one.

Each node has three main attributes:

- *type*: identifying the type of the node (primitive, +, &, |, etc.)
- *guard*: representing the predicate that must be evaluated when an event is detected
- *history*: an ordered history of *event records*

Every time a primitive event is triggered, the relevant leaves are informed of the occurrence. Each leaf node enters a corresponding event record in its history if its guard is satisfied and informs its parent node of the existence of the new record[6]. Depending on its semantics each parent in turn uses the new record from its child, its associated guard, and possibly the records in the histories of its other children (if any) to generate as many new event records as possible. The parent may generate and add to its history several such records. The process continues until the rule fires or no new records can be generated.

Each record may refer to one or more other records at lower level nodes and can be viewed as an *event instance tree*, the leaves of which refer to primitive event instances. The histories maintained by the event evaluation tree form a *hierarchical history* which avoids unnecessary orderings. Only related, rather than all events are ordered. This history is updated in a consistent way as new records are inserted. Duplication of event information within each rule is kept to a minimum as event records only maintain pointers to other event records. A guard evaluation tree is associated with each event evaluation node. This allows incremental evaluation of the guards and therefore early filtering of events which in turn reduces unnecessary computations. As we saw in section 5.2, one or more records may be invalidated as a result of late-arriving events and have to be cancelled. Cancellations may also occur as a result of rule firings as shown in the following example. The structure of the tree allows such cancellations to be dealt with easily and efficiently.

Dealing with delays at the detection stage adds to the complexity of the event evaluation tree and the detection algorithm. In addition to evaluation of event expressions, the algorithm must perform event ordering and cancelling invalidated partial results. This complexity is unavoidable for remote monitoring but unnecessary for monitoring events generated locally.

6.2 An Example of Composite Event Detection

Figure 12 (i) shows the event evaluation tree and its hierarchical history created for a rule of the form:

```
rule test_rule [11*hour] { (a & b) ; ({c ; d} ! e) ==> ... }
```

The notation e_t is used to refer to an event instance of class e with the occurrence timestamp t. Assume that each unit on the time line represents one hour and that the rule was enabled at 5am with a detection window of 11 hours. Figures 12 (i—vi) illustrate the changes in the hierarchical history as the sequence of event instances shown in figure 11 are detected.

---

6 Multiple event records in the histories of different leaf nodes (possibly belonging to various rules) may refer to the same event occurrence. Duplication is minimised by using pointers and reference counts.

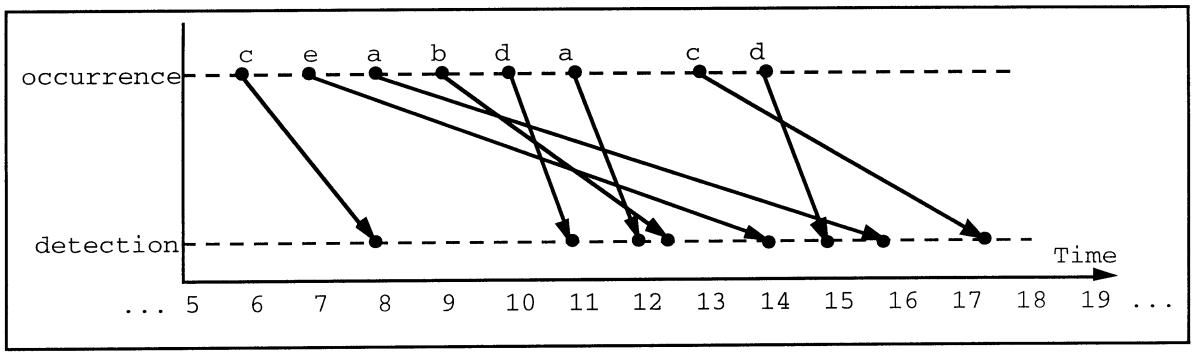

Figure 11 Occurrence and detection times of a delayed event sequence

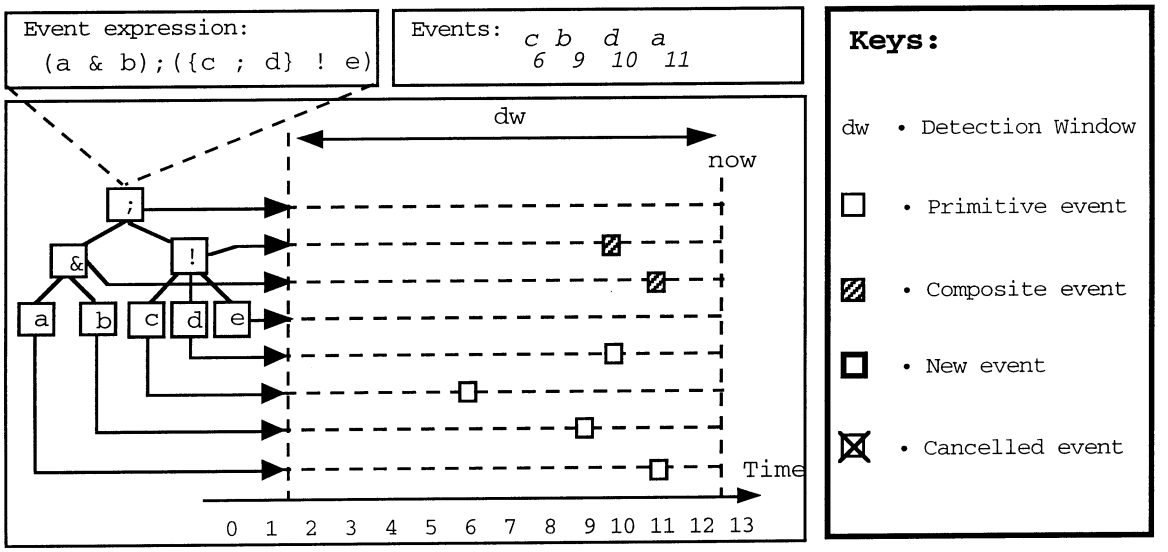

Figure 12 (i) Event evaluation tree

Figure 12 (i) shows the state of the event evaluation tree and its leaf nodes histories at time 13:00, after the detection of the following primitive events: $c_6\ b_9\ d_{10}\ a_{11}$. The figure also shows the event records (representing composite sub-events) which have been inserted in the histories of the corresponding internal nodes. As we have seen any event whose occurrence timestamp falls outside the rule's detection interval is ignored by the rule otherwise a corresponding event record is inserted in its correct place within the relevant history and can be used for evaluation.

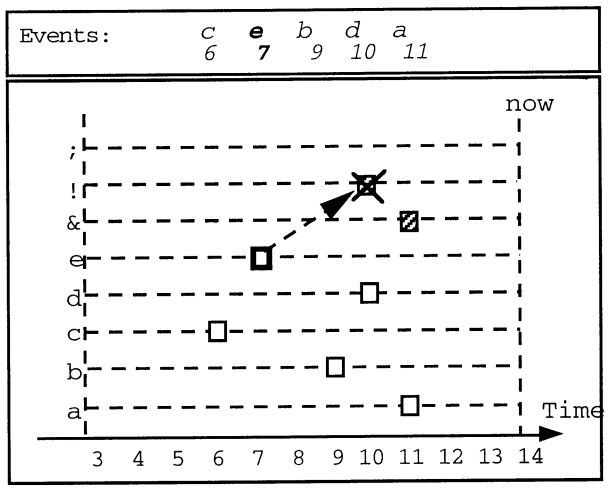

Figure 12 (ii)

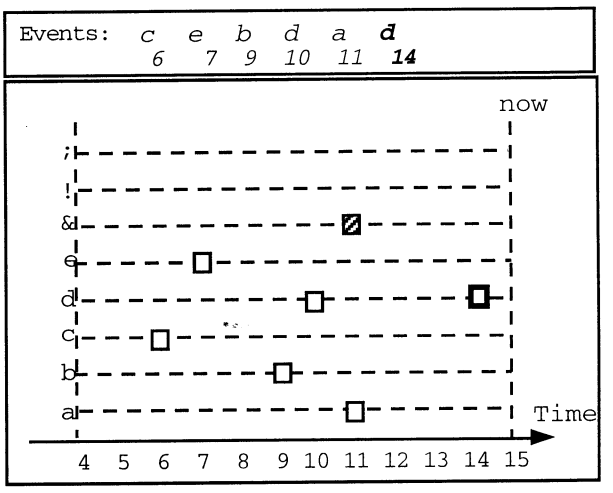

Figure 12 (iii)

At time 14:00 the event e_7 is detected (Figure 12 (ii)). Given the semantics of the '!' operator, it causes the composite event record ($\{c_6 ; d_{10}\} ! e$) to be cancelled. It illustrates one of the problems that delays introduce in the detection process. Some of the intermediate results may be invalidated due to late arriving events and would have to be *cancelled*. Any other higher level event records depending on the cancelled event would also have to be cancelled. At time 15:00 event d_{14} is detected (Figure 12 (iii)) and a corresponding event record is inserted in e_4's history. It does not result in the generation of any higher level event records.

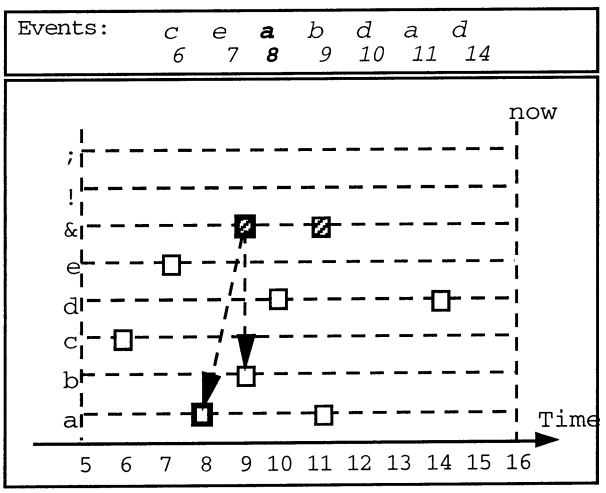

Figure 12 (iv)

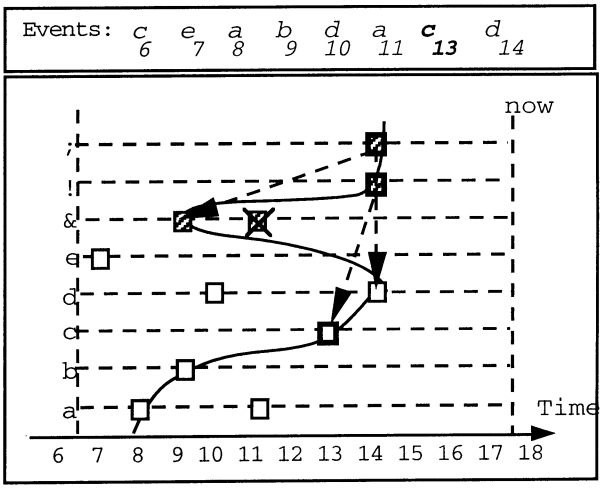

Figure 12 (v)

At time 16:00 event a_8 is detected which in turn causes the detection of the composite event (a_8 & b_9), as reflected in the hierarchical history in figure 12 (iv). Note that the new event records are inserted in their correct occurrence place in this history, irrespective of detection order.

Figure 12 (v) illustrates the detection of event c_{13} which has resulted in the detection of composite events at various levels, and in particular the occurrence of the top level composite event whose constituent primitive events are $(a_8, b_9, c_{13}, d_{14})$. The curved solid line is used to show records participating in this occurrence. This occurrence results in the firing of the rule and the execution of its action sequence, which may use the event information. Once all the actions are executed all the records contributing to this firing are marked for consumption. Consumed records will not be able to participate in any further evaluations and rule firings. As a result any higher level records which depend on consumed records would have to be cancelled (for example the crossed record in '&' node's history with timestamp 11, because it depends on the consumed event record representing b_9). Note that the detection algorithm selects events in a *chronological order* (e.g., choosing a_8 rather than a_{11}). As can be seen there may be no obvious relationship between the occurrence interval (8—14) and the detection interval (8 —17:30). A scheduled relative time event (a built-in feature of the language) is used as a unique terminator to link the two intervals and deliberately extend the detection interval to deal with late-arriving events. Figure 12 (vi) shows the state of the hierarchical history after event record consumption and cancellation.

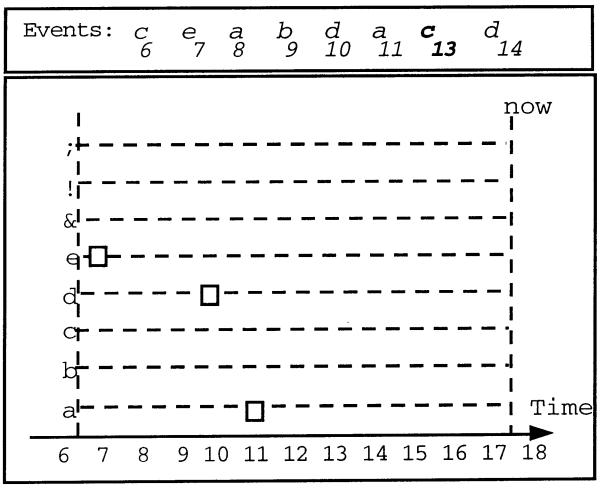

Figure 12 (vi)

7 RELATED WORK AND CONCLUSIONS

Event monitoring and in particular composite event detection has received much attention particularly in the area of active databases (Chakravarthy et al. 1993; Chakravarthy & Mishra 1993; Dayal 1988; Gehani et al. 1992). They provide rich notations for specifying composite events but due to their centralised nature the proposed detection algorithms are unsuitable for event composition in a loosely coupled distributed environment. In particular these algorithms assume that events are detected at the time or in the order that they occur. As we have seen such assumptions cannot be made in a distributed environment and therefore using such algorithms may lead to erroneous and alternative detections and hence unwanted rule firings.

Various mechanisms for composite event detection have been employed. Samos (Gatziu & Dittrich 1994) uses Petri Nets for event detection. Ode (Gehani et al. 1992) uses a finite automata. A similar mechanism is employed in Meta (Marzullo et al. 1991) for evaluating temporal expressions. The mechanism used by (Bacon et al. 1995) is based on finite state machines with enhancements to allow detection of concurrent composite events. We have adopted a tree-based mechanism as it seems to be the most suitable for on-the-fly detection of concurrent composite events in presence of delays. (Chakravarthy et al. 1993; Shim & Ramamoorthy 1990) use a similar mechanism, but in contrast to those our event evaluation tree maintains a *hierarchical* history which allows us to deal with out-of-order event arrivals in an efficient manner.

A number of systems use centralised correlation approach to filter and analyse events generated from distributed sources (Jakobson & Weissman 1995; Klinger et al. 1995; Nygate 1995). The problem with this approach is that all events have to be sent to the correlation server which can cause considerable network traffic in large systems. Our approach allows filtering and composition of events to be distributed and take place close to the source of these events and so it scales for very large systems. A similar approach has been adopted by (Spezialetti & Bernberg 1995) who provide the ability to define dynamic configurations of the monitoring system, but it is not certain how event detection is performed in the presence of delays.

Event-driven monitoring has been used extensively for observing and analysing the behaviour of distributed systems (Bates 1995; Haban & Wybranietz 1990; LeBlanc & Robbins 1985; Lumpp et al. 1990; Marzullo et al. 1991; Spezialetti & Kearns 1988; Wolfson et al. 1991). It is used for a variety of tasks such as debugging, performance analysis, program visualisation and management. See (Monsouri-Samani & Sloman 1994) for a survey of distributed systems monitoring. None of the current approaches provide a flexible and user-defined facility for dealing with delays when detecting composite events in a distributed environment.

GEM provides a generalised event monitoring notation that permits user specified filtering and composition scripts to be dynamically loaded into distributed event monitoring components. GEM uses scheduled time events and default or user defined detection windows to cope flexibly with the problems that variable communication delays introduce in detecting composite events. The GEM monitor has been implemented in C++ within the REGIS/DARWIN environment used for communication and configuration (Magee et al. 1994). Bison++ and Flex++ have been used to implement the interpreter.

The GEM event monitoring system is being used to detect complex event sequences and convert these into simple events which trigger obligation policies within managers which then perform the management actions specified by these activities (Sloman 1994). The event monitors have therefore been restricted to performing very simple activities related to triggering or notifying events. We have separated the event handling from the management which is in contrast to the approach taken by the triggered databases which combine event handling and database activities into a single component.

ACKNOWLEDGEMENTS

We gratefully acknowledge EPSRC and HP Laboratories Bristol for a CASE studentship as well as financial support form Esprit SysMan (7026) Project . We also acknowledge the contribution of our colleagues, to the concepts discussed in this paper.

REFERENCES

Bacon, J., Bates, J., Richard, H. & Moody, K. (1995) *Using Events to Build Distributed Applications,* In Proceedings of the 2nd International Workshop on Services in Distributed and Network Environments, Whistler, British Colombia, pp. 148—155.

Bates, P. (1995) *Debugging Heterogeneous Distributed Systems Using Event-Based Models of Behaviour*, ACM Transactions on Computer Systems, Vol. 13, No. 1, pp. 1—31.

Brownston, L., Farrell, R., Kant, E., Martin, N. (1985) *Programming Expert Systems in OPS5: An Introduction to Rule-Based Programming*: Addison Wesley.

Chakravarthy, S., Krishnaprasad, V., Anwar, E. & Kim, S.-K. (1993) *Anatomy of a Composite Event Detector*, Technical Report UF-CIS-TR-93-039, University of Florida, Department of Computer and Information Sciences.

Chakravarthy, S. & Mishra, D. (1993) *Snoop: An Expressive Event Specification Language For Active Databases*, Technical report UF-CIS-TR-93-007, University of Florida, Department of Computer and Information Sciences.

Christian, F. (1989) *Probabilistic Clock Synchronisation*, Distributed Computing, Vol. 3, pp. 146—158.

Dayal, U. (1988) *Active Database Management Systems,* In Proceedings of the 3rd International Conference on Data and Knowledge Bases, Jerusalem, pp. 150—169.

Gatziu, S. & Dittrich, K. R. (1994) *Detecting Composite Events in Active Databases using Petri Nets,* In Proceedings of the 14th International Workshop on Research Issues in data Engineering: Active Database Systems, Houston, Texas.

Gehani, N., Jagadish, H. V. & Shmueli, O. (1992) *Composite Event Specification in an Active Databases: Model & Implementation,* In Proceedings of the 18th VLDB Conference, Vancouver, British Colombia, Canada.

Haban, D. & Wybranietz, D. (1990) *A Hybrid Monitor for Behaviour and Performance Analysis of Distributed Systems*, IEEE Transactions on Software Engineering, Vol. 16, No. 2.

Jakobson, G. & Weissman, M. (1995) *Real-time Telecommunication Network Management: Extending Event Correlation with Temporal Constraints,* In Proceedings of the Fourth Symposium on Integrated Network Management, Santa Barbara, California, USA, Chapman & Hall, pp. 290—301.

Klinger, S., Yemini, S., Yemini, Y., Ohsie, D. & Stolfo, S. (1995) *A Coding Approach to Event Correlation,* In Proceedings of the Fourth Symposium on Integrated Network Management, Santa Barbara, California, USA, Chapman & Hall, pp. 290—301.

LeBlanc, R. J. & Robbins, A. D. (1985) *Event-Driven Monitoring of Distributed Programs,* In Proceedings of the 5th International Conference on Distributed Computing Systems, pp. 515—522.

Lumpp, J. E., Jr., Casavant, T. L., Seigle, H. J. & Marinescu, D. C. (1990) *Specification and Identification of Events for Debugging and Performance Moitoring of Distributed Multiprocessor Systems,* In Proceedings of the 10th International Conference on Distributed Systems, pp. 476—483.

Magee, J., Dulay, N. & Kramer, J. (1994) *REGIS: A Constructive Development Environment for Distributed Programs*, IOP/IEE/BCS Distributed Systems Engineering, Vol. 1, No. 5, pp. 304—312.

Marzullo, K., Cooper, R., Wood, M. D. & Birman, K. P. (1991) *Tools for Distributed Application Management*, IEEE Computer, Vol. 24, No. 8, pp. 42—51.

Monsouri-Samani, M. & Sloman, M. (1994) *Monitoring Distributed Systems*, In M. Sloman (Ed.), Network and Distributed Systems Management, Addison Wesley, pp. 303—347.

Nygate, Y. A. (1995) *Event Correlation using Rule and Object Based Techniques,* In Proceedings of the Fourth Symposium on Integrated Network Management, Santa Barbara, California, USA, Chapman & Hall, pp. 290—301.

Shim, Y. C. & Ramamoorthy, C. V. (1990) *Monitoring and Control of Distributed Systems,* In Proceedings of the First International Conference on Systems Integration, Morristown, NJ, IEEE Computing Press, pp. 672—681.

Sloman, M. (1994) *Policy Driven Management for Distributed Systems*, Plenum Press Journal of Network and Systems Management, Vol. 2, No. 4, pp. 333—360.

Spezialetti, M. & Bernberg, S. (1995) *EVEREST: An Event Recognition Testbed,* In Proceedings of the 15th International Conference on Distributed Computing Systems, Vancouver, IEEE Computing Society, pp. 377—385.

Spezialetti, M. & Kearns, J. P. (1988) *A General Approach to Recognising Event Occurrences in Distributed Computations,* In Proceedings of the 8th International Conference on Distributed Computing Systems, pp. 300—307.

Wolfson, O., Sengupta, S. & Yemini, Y. (1991) *Managing Communication Networks by Monitoring Databases*, IEEE Transactions on Software Engineering, Vol. 17, No. 9, pp. 944—953.